DYNAMICALLY GENERATED WRAPPER

INVENTORS
Fei Luo
Alexander J. Somogyi
William John Gallagher
Rahul Srivastava

CERTIFICATE OF MAILING BY "EXPRESS MAIL"
UNDER 37 C.F.R. §1.10

"Express Mail" mailing label number: EV 327 622 903 US
Date of Mailing: November 12, 2003

I hereby certify that this correspondence is being deposited with the United States Postal Service, utilizing the "Express Mail Post Office to Addressee" service addressed to Mail Stop Patent, Commissioner for Patents, P.O. Box 1450, Alexandria, Virginia 22313 and mailed on the above Date of Mailing with the above "Express Mail" mailing label number.

_____(Signature)
Tina Galdos
Signature Date: November 12, 2003

DYNAMICALLY GENERATED WRAPPER

INVENTORS:
Fei Luo
Alexander J. Somogyi
William John Gallagher
Rahul Srivastava

Claim to Priority

[0001] The present application claims the benefit of priority under 35 U.S.C. §119(e) to U.S. Provisional Patent Application entitled "DYNAMICALLY GENERATED WRAPPER", Application No. 60/450,901, filed on February 28, 2003, which application is incorporated herein by reference.

Copyright Notice

Cross Reference to Related Applications

[0003] The present application is related to the following United States Patents and Patent Applications, which patents/applications are assigned to the owner of the present invention, and which patents/applications are incorporated by reference herein in their entirety:

[0004] United States Patent Application No. 10/XXX,XXX, entitled "DYNAMIC CODE GENERATION SYSTEM", filed on XXX XX, 2003, attorney reference number BEA-1316US2; and

[0005] United States Patent Application No. 10/XXX,XXX, entitled "METHOD FOR DYNAMICALLY GENERATING A WRAPPER", filed on XXX XX, 2003.

Field of the Invention

[0006] The current invention relates generally to wrapping software objects, and more particularly to dynamically wrapping software objects.

Background of the Invention

[0007] Application servers provide an environment for programmers to implement application programs for users. BEA System of San Jose, California, provides one such application server called Web Logic Server (WLS). Third party vendors typically provide resource adapters, JDBC, JMS and JCA drivers, for use on these application servers. A programmer may use those resource adapters to provide the services to the user through the programmer's application program. Vendor resource adapters are varied and generally provide a broad range of features to be accessed by application programs.

[0008] Within an application server, it is desirable to provide support for global transactions. Generally, this involves tracking the status and availability of resources, changes to shared resources throughout the servers, and managing other features that affect the server system. Accordingly, an application server should be informed regarding transactions that occur between application programs and resources. To effectuate this, an application server may not provide application programs with direct access to vendor objects. Providing application programs with direct access to vendor objects would prevent the application server from knowing what was transpiring between application programs and resources. Application servers implement a middle-ware between the application programs and the resources to determine what transpires in transactions between them.

[0009] Previously, middle-ware was implemented by application servers in the form of statically generated proxies. The statically generated proxy included hard coded methods that handled interfaces for specific vendor resource adapters. The application server was designed with a proxy that assumed the presence of certain vendor features, such as a particular type of resource or a particular type of resource interface. These application servers worked well to provide the application programs access to those specific resources. However, an application program encountered difficulties if it was configured to use a resource adapter from a different vendor or any other resource adapter not statically included at the development time of the application server.

[0010] What is needed is a system and method for providing application programs access to extension features of third-party resource adapters in addition to allowing application server to monitoring the activities between application programs and resource adapters. It would be desirable to provide the full access to the vendor object to the application programs as well as

intercept calls made to and returned from the vendor object to handle server side tasks, global transaction enlistment, connection pooling, statement caching, tracing, and profiling.

Summary of the Invention

[0011] A wrapper object for a vendor object is generated dynamically at runtime. In one embodiment of the present invention, the Dynamic Generated Wrapper (DGW) is a proxy generated at runtime and acts as a delegate for the underlying vendor object being proxied. In one embodiment, the wrapper class may be a subclass of a statically predefined superclass that includes the programming logic to handle server side tasks. The wrapper class may include methods in the vendor class that are not present in the superclass. A wrapper object is an instance of a wrapper class. The wrapper object may be used to intercept method invocations from an application program to an vendor object and provide the execution of server side tasks in a pre-invocation handler and post-invocation handler. Additionally, the wrapper object may be used to intercept the result of the method invocation against vendor object. The wrapper object may provide for server side tasks to be performed before providing the wrapped result to the application programs. The server side tasks may include global transaction enlistment, resource pooling, resource caching, tracing and profiling.

Brief Description of the Drawings

[0012] FIGURE 1 is an illustration of a system for wrapping a vendor object in accordance with one embodiment of the present invention.

[0013] FIGURE 2 is an illustration of an expanded superclass in accordance with one embodiment of the present invention.

[0014] FIGURE 3 is an illustration of a method for dynamically generating a wrapper object in accordance with one embodiment of the present invention.

[0015] FIGURE 4 is an illustration of a workflow of a dynamically generated wrapper method in accordance with one embodiment of the present invention.

Detailed Description

[0016] A wrapper object for a vendor object is generated dynamically at runtime. In one embodiment of the present invention, the Dynamic Generated Wrapper (DGW) is a proxy generated at runtime and acts as a delegate for the underlying vendor object being proxied. In one embodiment, the wrapper class may be a subclass of a statically predefined superclass that includes the programming logic to handle server side tasks. The wrapper class may include methods in the vendor class that are not present in the superclass. The wrapper object may be used to intercept method invocations from an application program to the vendor object and provide the execution of server side tasks in a pre-invocation handler and post-invocation handler. Additionally, the wrapper object may be used to intercept the result of the method invocation against the vendor object. The wrapper object may provide for server side tasks to be performed before providing the wrapped result to the application program . The server side tasks may include global transaction enlistment, resource pooling, resource caching, tracing and profiling.

[0017] In one embodiment, it is desirable for application servers that host enterprise applications to allow application programs access to Java Database Connectivity (JDBC) vendor extensions. Accordingly, the wrapper object of the present invention may be implemented as a JDBC wrapper object that acts as a proxy between application programs and JDBC vendor objects. To implement application program access to vendor extension methods, JDBC wrapper class should implement vendor extension interfaces. If the JDBC wrapper class of the present invention does not implement an vendor extension interface, application programs may not utilize the vendor extension features.

[0018] In an application server, an application program may invoke a method against a vendor

object. In one embodiment of the present invention, a dynamically generated wrapper is used to intercept transmission between vendor object and application program, initiate processing related to the transmissions, and provide the wrapped result back to the application program. FIG. 1 illustrates a system 100 for providing a dynamic generated wrapper for a vendor object in an application server in accordance with one embodiment of the present invention. FIG. 1 includes vendor object 110, dynamic generated wrapper object 120 and application program 130. The dynamic generated wrapper object 120 intercepts communications between the vendor object 110 and the application program 130 as shown. The dynamic generated wrapper object may perform processing at different stages of the invocation of method.

[0019] FIG. 2 illustrates a system 200 having a dynamically generated wrapper class 210 that extends from a statically predefined superclass 220. Dynamically generated wrapper 210 also implements all of the interfaces, including vendor extension interfaces 240 and J2EE interfaces 250, supported by vendor class 230. As discussed above, a vendor class may have a large number of extension methods. These extension methods can be divided into two groups. The first group of those methods are implemented in a superclass. They are known to the application server. The wrapper class is a sub-class of the superclass. Thus, every method in the superclass is also contained in the wrapper class.

[0020] A typical vendor class may contain a large number of methods. Within these methods, few of them may require special treatment. The special treatment may differ from the typical processing that can be applied to the rest of methods with respect to server side tasks. The remainder of the methods may be handled with a wrapper class in a standard manner such as that illustrated in FIG. 3. For the methods requiring special treatment, a wrapper method should not be automatically generated. These special treatment methods are contained in the statically

predefined superclass.

[0021] In one embodiment, the superclass 220 has several requirements. In one embodiment, the superclass must be public and not final. Thus, the superclass may be extended by the dynamically generated wrapper class. The superclass must implement a predefined wrapper interface that includes the preInvocation and postInvocation handler. The superclass requires a member variable, to hold the proxied vendor object, and a non-arg constructor, to instantiate the wrapper object. The superclass should also have an init method for initializing the generated wrapper object.

[0022] Below is an example of what code implementing a super class may look like.

```
package weblogic.jdbc.pool;
import weblogic.utlis.wrapper.*;

/* WrapperImpl implements Wrapper interface */
public class Connection extends WrapperImpl {
// override the super class method
public Object postInvocationHandler(String methodName, Object[] params,
Object ret) {
        super.postInvocationHandler(methodName, params, ret);
        System.out.println("Doing Pooling Stuff");
        return ret;
}
.....
}
```

An example of the interface that a superclass implements is shown below.

```
public interface Wrapper {
/**
        set vendor object proxied by this wrapper
* @param obj                  vendor object
*/
public void   setVendorObj(Object obj);
/**
        get vendor object proxied by this wrapper
* @return              vendor object
*/
public Object getVendorObj();
/**
        this wrapper method gets invoked before vendor object gets invoked
        @param methodName -name of method that will be invoked on the vendor
        object
        @param params -array of parameters that will be passed to the vendor
```

```
        object
*/
public void preInvocationHandler(String methodName, Object[] params)
        throws Exception;
/**
        this wrapper method gets invoked after the vendor object gets
        invoked
* @param methodName     name of method that was invoked on vendor object
        @param params-array of parameters that was passed to vendor object
* @param ret      -return value that was returned by vendor object
* @return   -return value that will be return to the end user
*/
public Object postInvocationHandler(String methodName, Object[] params,
Object ret)
        throws Exception;
}
```

[0023] FIG. 3 illustrates a method 300 for dynamically generating a wrapper object in accordance with one embodiment of the present invention. The wrapper object may be dynamically generated at runtime. The method may be performed by a "wrapper factory" software residing in the application server. Method 300 begins with start step 305. Next, the vendor object and superclass are received in step 310. In one embodiment, the superclass may be one of either pre-existing WLS JDBC, WLS JMS or WLS Connector classes. The pre-existing superclass is used to generate its subclass, wrapper class. In one embodiment, the superclass may contain logic to handle server side tasks including global transaction management, pooling, caching, tracing and profiling.

[0024] Next, the system performs reflection on the vendor class at step 320. Reflection is an operational feature of Java that allows meta information, what interfaces vendor class implemented, to be retrieved from code. The retrieved meta information allows the application server to dynamically generate a wrapper class, or proxy class, that perfectly matches with the vendor class.. The wrapper class is then generated in memory at step 330. The wrapper class is generated in byte code and extends from the superclass received at step 310. Code is generated for vendor methods not implemented in the superclass. In one embodiment of the present

invention, the generated code may be similar to the following code.

```
public int getTransactionIsoloation() throws SQLException{
preInvocationHandler("getTransactionIsoloation",null);

return postInvocationHandler(
"getTransactionIsoloation";
null,
vendorObj.getTransactionIsolation()
);
}
```

[0025] In one embodiment, the code is generated using hot code generation techniques as described in related United States Patent Application No. XX/XXX,XXX, entitled "Hot Code Generation", herein incorporated by reference in its entirety. After generating the wrapper class, an instance of the wrapper class, wrapper object, is created at step 340. The vendor object is then associated with the wrapper object at step 350. The returned wrapper object is then provided to the application program such that the application program may access both the standard features and non-standard vendor extensions. In one embodiment, the standard features are J2EE features. Operation of method 300 then ends at step 355. The wrapper class includes all public interfaces implemented by the vendor class required by the application program. As a result, the application program may cast the wrapper object to the vendor interface to access vendor extension methods.

[0026] The application server system may have code for generating the wrapper. An example of code for generating the wrapper is below.

```
package weblogic.jdbc.Driver;
import weblogic.utlis.wrapper.*;

public class Driver implements java.sql.Driver {
.....
public java.sql.Connection connect(String url, Properties dbprops)
        throws SQLException
{
        .....
        ConnectionEnv  cc  =  ConnectionPool.reserveWaitSecs(subject,  poolID,
```

```
        appName, waitSecs);

        /* current code with hardcoded wrapper */
        // Connection poolConnection = new Connection(cc);
        /* current code with hardcoded wrapper */

        /* new code with WDGW */
        Connection poolConnection = /* wrapper object */
        (Connection)WrapperFactory.getWrapper(
               "weblogic.jdbc.pool.Connection", /*super class */
               cc.conn.jconn, /* vendor object */
               false
        );
        poolConnection.init(cc); /* initialize the wrapper object */
        /* new code with WDGW */

        return (java.sql.Connection)poolConnection;
}
.....
}
```

[0027] Once the wrapper object is dynamically created from the vendor object and superclass, the wrapper object may be used to wrap the particular vendor object. FIG. 4 illustrates a method 400 for using a dynamic wrapper object for processing a method invocation against a vendor object in accordance with one embodiment of the present invention. Method 400 begins with start step 405. Next, a call is received by the wrapper object at step 410. The call is made by the application program to the wrapped vendor object. The wrapper object may then initiate any pre-processing to be performed at step 420. In one embodiment, the pre-processing includes calling a pre-invocation handler. The pre-invocation handler may be configured to execute server-side code before the vendor methods are invoked. Application server code to be executed before the vendor code invocation may include global transaction processing.

[0028] After any pre-processing is performed, a call is made to the wrapped vendor object at step 430. The wrapper object forwards the call to the vendor object on behalf of the application program. The result of the vendor object call is then received by the wrapper object at step 440. The wrapper object may then initiate any post-processing to be performed at step 450. In one embodiment, post-processing may include calling a post-invocation handler. The post-

invocation handler may be used to perform post-processing server-side tasks including global transaction management and wrap the returned result with another wrapper. At step 460, the wrapped result is provided to the application program. Operation of method 400 then ends at step 465.

[0029] In one embodiment of the present invention, a static method may be used to generate a wrapper for a vendor object. An example of code implementing the static method in WLS is shown below.

```
package weblogic.utils.wrapper;
public class WrapperFactory {
/**
         Get a wrapper for a vendor object
*
* @param superClassName name of supper class of the wrapper class
* @param vendorObj            vendor object proxied by returned wrapper
* @param remote         whether generate a RMI remote wrapper
*
* @return               wrapper
*/
public final static Object getWrapper(
         String superClassName,
         Object vendorObj,
         boolean remote);
}
```

[0030] Other features, aspects and objects of the invention can be obtained from a review of the figures and the claims. It is to be understood that other embodiments of the invention can be developed and fall within the spirit and scope of the invention and claims.

[0031] The foregoing description of preferred embodiments of the present invention has been provided for the purposes of illustration and description. It is not intended to be exhaustive or to limit the invention to the precise forms disclosed. Obviously, many modifications and variations will be apparent to the practitioner skilled in the art. The embodiments were chosen and described in order to best explain the principles of the invention and its practical application,

thereby enabling others skilled in the art to understand the invention for various embodiments and with various modifications that are suited to the particular use contemplated. It is intended that the scope of the invention be defined by the following claims and their equivalence.

[0032] In addition to an embodiment consisting of specifically designed integrated circuits or other electronics, the present invention may be conveniently implemented using a conventional general purpose or a specialized digital computer or microprocessor programmed according to the teachings of the present disclosure, as will be apparent to those skilled in the computer art.

[0033] Appropriate software coding can readily be prepared by skilled programmers based on the teachings of the present disclosure, as will be apparent to those skilled in the software art. The invention may also be implemented by the preparation of application specific integrated circuits or by interconnecting an appropriate network of conventional component circuits, as will be readily apparent to those skilled in the art.

[0034] The present invention includes a computer program product which is a storage medium (media) having instructions stored thereon/in which can be used to program a computer to perform any of the processes of the present invention. The storage medium can include, but is not limited to, any type of disk including floppy disks, optical discs, DVD, CD-ROMs, microdrive, and magneto-optical disks, ROMs, RAMs, EPROMs, EEPROMs, DRAMs, VRAMs, flash memory devices, magnetic or optical cards, nanosystems (including molecular memory ICs), or any type of media or device suitable for storing instructions and/or data.

[0035] Stored on any one of the computer readable medium (media), the present invention includes software for controlling both the hardware of the general purpose/specialized computer or microprocessor, and for enabling the computer or microprocessor to interact with a human user or other mechanism utilizing the results of the present invention. Such software may

include, but is not limited to, device drivers, operating systems, and user applications.

[0036] Included in the programming (software) of the general/specialized computer or microprocessor are software modules for implementing the teachings of the present invention, including, but not limited to, dynamically generating a vendor object wrapper.